Compilation – TD1 : Traduction dirigée par la syntaxe

Nawfal ‘Massine’ MALKI – 4991 – STI 3A TD3

Exercice 1 : Machine à pile

Programme 1

```
alloc 2        alloue 2 slots
push 0         empile le contenu x du slot 0 dans la pile
push 1         empile le contenu y du slot 1 dans la pile
add            additionne les contenus des slots 1 et 0 (x et
               y) et les empile, appelons ce résultat z
mpush 1        empile la valeur immédiate 1
sub            soustrait 1 de z
pop 0          dépile z-1 dans le slot 0
free           libère les slots alloués (0 et 1)
```

Résultat

```
int x,y ;
x = x+y-1;
```

Programme 2

```
alloc 2        alloue 2 slots
push 0         empile le contenu x du slot 0 dans la pile
push 1         empile le contenu y du slot 1 dans la pile
testg          test x > y
jz end         si faux, on saute au label end, si vrai :
push 0         empile le contenu x du slot 0 dans la pile
push 1         empile le contenu y du slot 1 dans la pile
sub            on effectue x – y et on l’empile
pop 0          on dépile dans le slot 0
end:free       libère les slots alloués
```

Résultat

```
int x,y;
if (x > y){
     x = x-y;
}
```

Exercice 2 : Traduction dirigée par la syntaxe

1. Proposez une règle de traduction pour le while.

```
[[while(C) S]] =
     while :
          [[C]]
          jz end:
          [[S]]
          jmp while
     end :
```

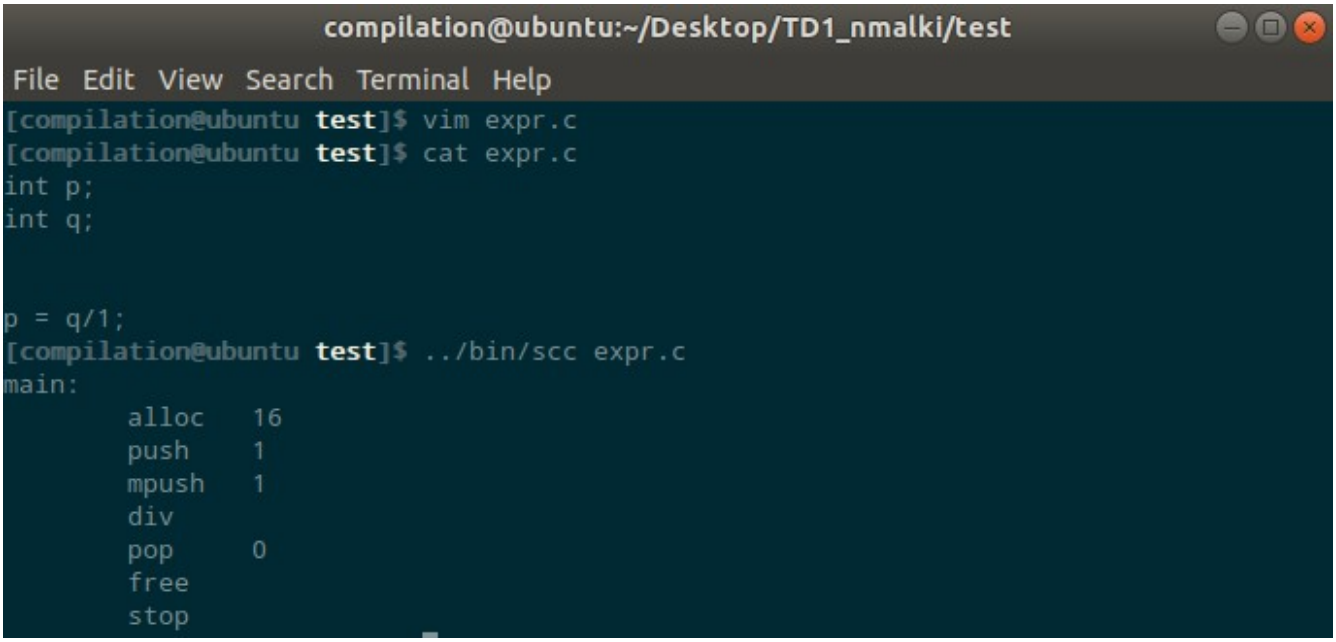
2. Traduire le programme en déroulant les règles de traduction.

```
alloc 2
while_:
     push 0
     push 1
     testne
     jz end
     push 0
     push 1
     testg
     jz else
     sub
     pop 0
     jmp while
else_:
     push 1
     push 0
     sub
     pop 1
     jmp while
end :
     free
     stop
```

Exercice 3 : Implémentation

Non-terminal	Catégorie syntaxique	Exemple
Expr	Expressions	`2*x + 1`
Test	Condition	`2*x+y!=0`
Stmt	Statement	`If, while`
declare_local_vars	Variable declaration	`Int x,y;`
prog	Main production	`program=declarations+statements`

Expressions (expr.c) : division



Tests (test.c) : stricte inégalité, différent de

```
[compilation@ubuntu test]$ cat test.c
int p;

if(p>0) p = 2*p;
[compilation@ubuntu test]$ ../bin/scc test.c
main:
        alloc   8
        push    0
        mpush   0
        testg
        jz      else_0
        mpush   2
        push    0
        mul
        pop     0
        jmp     endif_0
else_0:
endif_0:
        free
        stop
[compilation@ubuntu test]$ vim test.c
[compilation@ubuntu test]$ cat test.c
int p;

if(p!=0) p = 2*p;
[compilation@ubuntu test]$ ../bin/scc test.c
main:
        alloc   8
        push    0
        mpush   0
        testne
        jz      else_0
        mpush   2
        push    0
        mul
        pop     0
        jmp     endif_0
else_0:
endif_0:
        free
        stop
[compilation@ubuntu test]$
```

While (pgcd.c)

```
[compilation@ubuntu test]$ ../bin/scc pgcd.c
main:
        alloc   16
        call    input_int
        push    rax
        pop     0
        call    input_int
        push    rax
        pop     1
while_0:
        push    0
        push    1
        testne
        jz      end_while_0
        push    0
        push    1
        testg
        jz      else_0
        push    0
        push    1
        sub
        pop     0
        jmp     endif_0
else_0:
        push    1
        push    0
        sub
        pop     1
endif_0:
        jmp     while_0
end_while_0:
        push    0
        call    print_int
        call    print_newline
        free
        stop
```

Génération de code pour architecture x86-64

On modifie le Makefile avec le backend correspondant, on compile avec les exemples simple.out et pgcd.out et on obtient bien un binaire qui prend en compte les entrées/sorties.

```
[compilation@ubuntu test]$ make simple.out
../bin/scc simple.c
nasm -f elf64 simple.asm
gcc -no-pie -o simple.out simple.o
rm simple.o simple.asm
[compilation@ubuntu test]$ ./simple.out
4
5
[compilation@ubuntu test]$ make pgcd.out
../bin/scc pgcd.c
nasm -f elf64 pgcd.asm
gcc -no-pie -o pgcd.out pgcd.o
rm pgcd.asm pgcd.o
[compilation@ubuntu test]$ ./pgcd.out
8
4
4
```

On retrouve les implémentations des fonctions input() et print() dans le fichier bison parser.ypp. Et leurs tokens respectifs dans l'analyseur lexical lexer.l.

Dans parser.ypp :

```
194 /* ---                          I/O                                -- */
  1
  2 | TK_PRINT TK_LPAR expr TK_RPAR TK_SEMI
  3     {
  4       call_runtime("print_int");
  5       call_runtime("print_newline");
  6     }
  7
  8 | TK_INPUT TK_LPAR TK_ID TK_RPAR TK_SEMI
  9     {
 10       call_runtime("input_int");
 11       raw("\tpush\trax");
 12       pop(get_address($3));
 13     }
```